Solving ACiD-WG Design Problems with Petri Net Based Methods

Alex Yakovlev (alex.yakovlev@ncl.ac.uk)
Department of Computing Science,
University of Newcastle upon Tyne, England
(with support of Petrify group and Newcastle team)

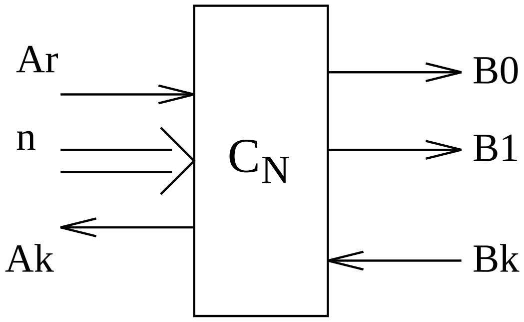
ACiD-WG Workshop – Groningen – 10 September 1996

Outline

- The Loadable Counter:
  Specification; Structuring the solution - basic cells; Abstract Petri net models; Signal Transition Graphs; Logic implementation.

- Interrupt Handler: Specification; Structuring the solution; Signal Transition Graphs; Logic implementation.

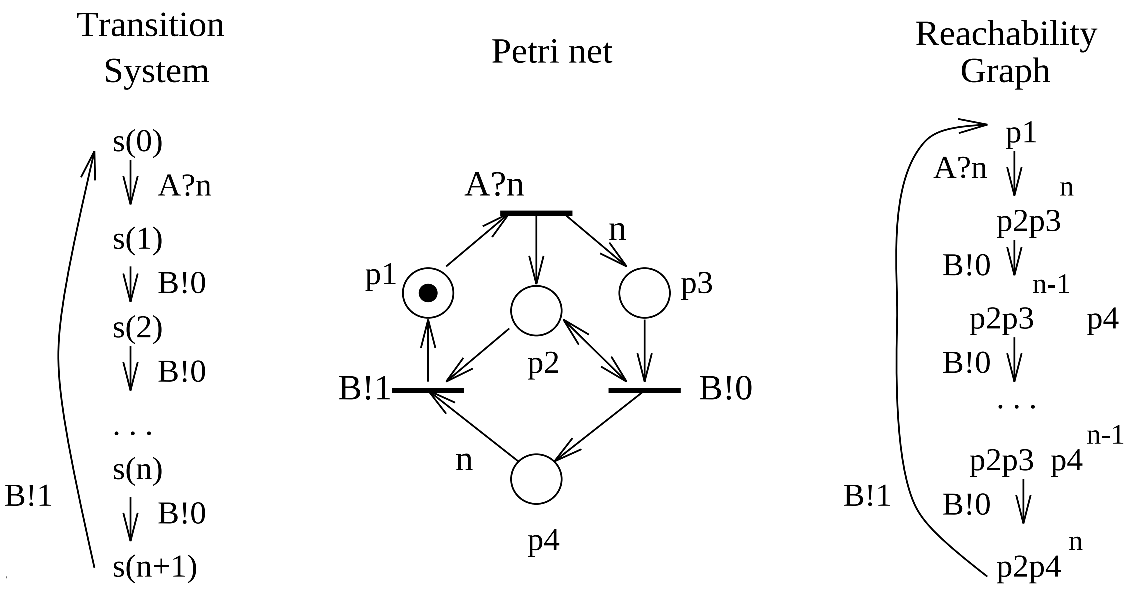
Loadable Counter: Original Specification



$(A?n;(B!0)^n;B!1)^*$

$$0 \leq n \leq N \leq 2^{k+1}, k = \lfloor \log N \rfloor$$

Towards Petri Net Specification



Representation of n

$$n = ((\ldots((0*2+n[k])*2+n[k-1])*2+\ldots+n[1])*2+n[0]),$$

where:
$n[k]$ is Most Significant Bit (MSB)
$n[0]$ is Least Significant Bit (LSB)
E.g. $k = 3$:

$$\begin{aligned} 5(0101) &= ((((0*2+0)*2+1)*2+0)*2+1) \\ 11(1011) &= ((((0*2+1)*2+0)*2+1)*2+1) \end{aligned}$$

Representation for $k = 2$

	bin	2		1		0	
0	000	*2	+0	*2	+0	*2	+0
1	001	*2	+0	*2	+0	*2	+1
2	010	*2	+0	*2	+1	*2	+0
3	011	*2	+0	*2	+1	*2	+1
4	100	*2	+1	*2	+0	*2	+0
5	101	*2	+1	*2	+0	*2	+1
6	110	*2	+1	*2	+1	*2	+0
7	111	*2	+1	*2	+1	*2	+1

For each Bit we have a Pair:

- Left Part is "*2"
- Right Part is "$+n[i]$", i.e. either "+0" or "+1"

Introducing "Skip Stages"

Series of operations "*2" and "+1":

$$(\ldots\underbrace{(\ldots((0*2+0)*2+0)*2+\ldots+0)}_{i\ \text{bits}}*2+\underbrace{1}_{n[k-i]})*2+\ldots+n[0])$$

can be reduced by setting all i bits of greater significance than $n[k-i]$ to special *Spacer* ("no work") state and generating counting only from the *highest nonzero* bit $n[k-i]$:

$$(\ldots(\underbrace{0}_{Part\text{"0"}}+\underbrace{1}_{n[k-i]})*2+\ldots+n[0])$$

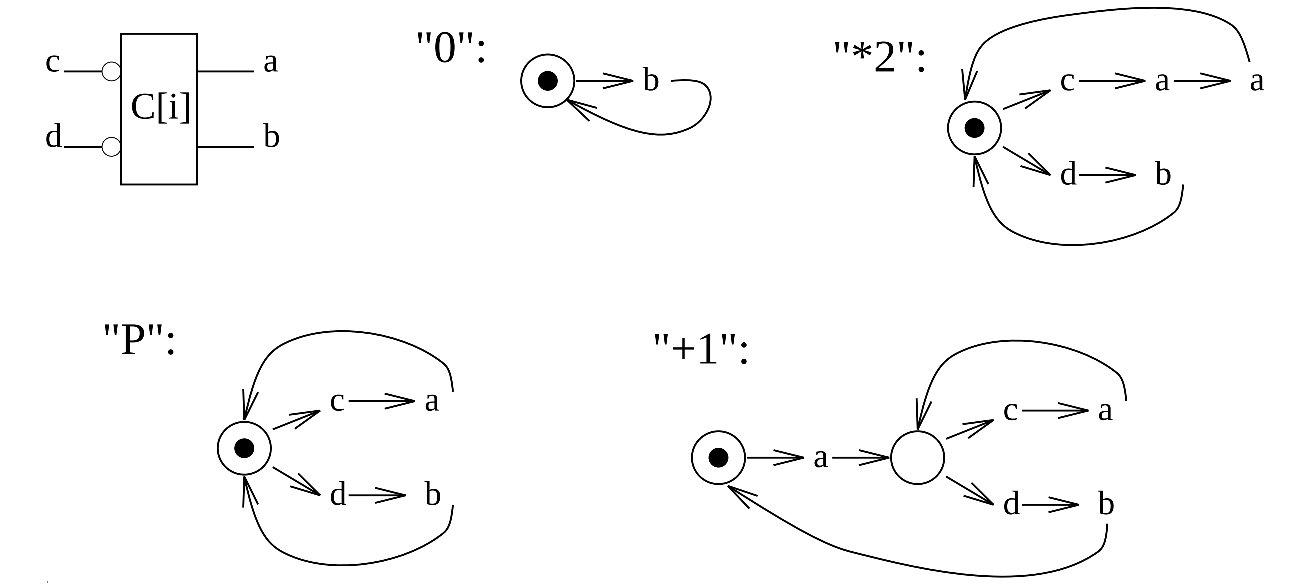
Cell Representation for $k=2$

	bin	2		1		0	
0	000	-	-	-	-	0	P
1	001	-	-	-	-	0	+1
2	010	-	-	0	+1	*2	P
3	011	-	-	0	+1	*2	+1
4	100	0	+1	*2	P	*2	P
5	101	0	+1	*2	P	*2	+1
6	110	0	+1	*2	+1	*2	P
7	111	0	+1	*2	+1	*2	+1

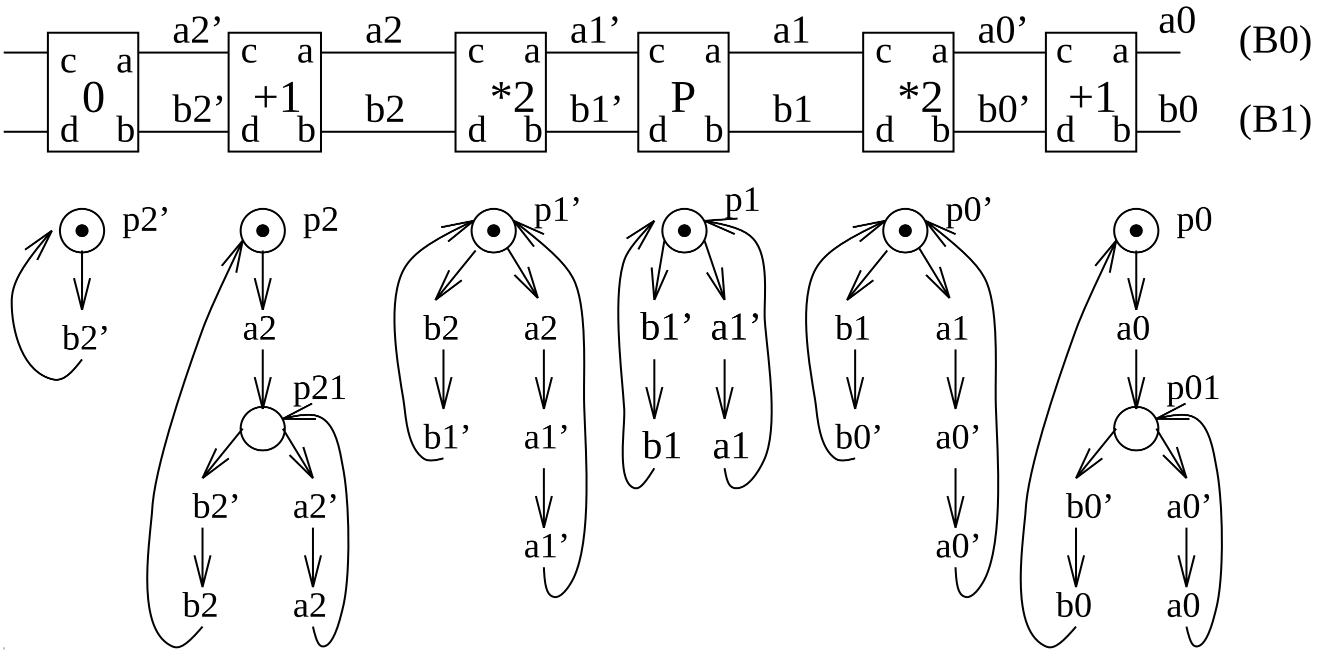
Types of Cells:

- "-" is *Spacer*, no count all
- "0" is *Zero*, count 0 times
- "*2" is *Even*, count $n' * 2$ times
- "P" is *Pass*, count n' times
- "+1" is *Odd*, count $n' + 1$ times

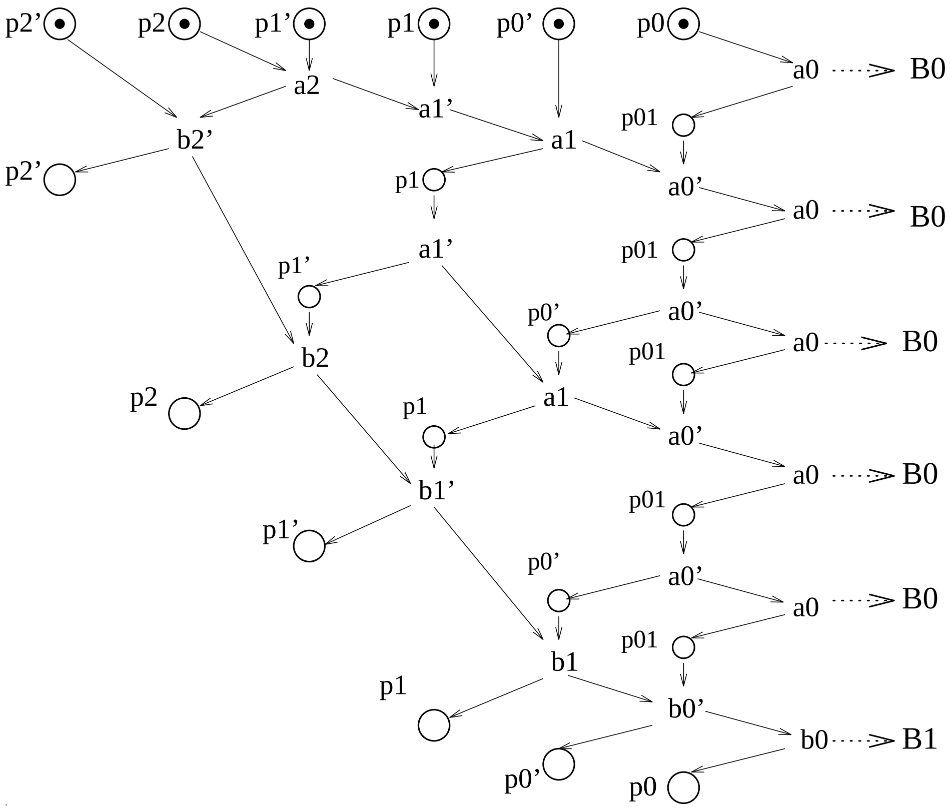
Petri Nets for Cell Types



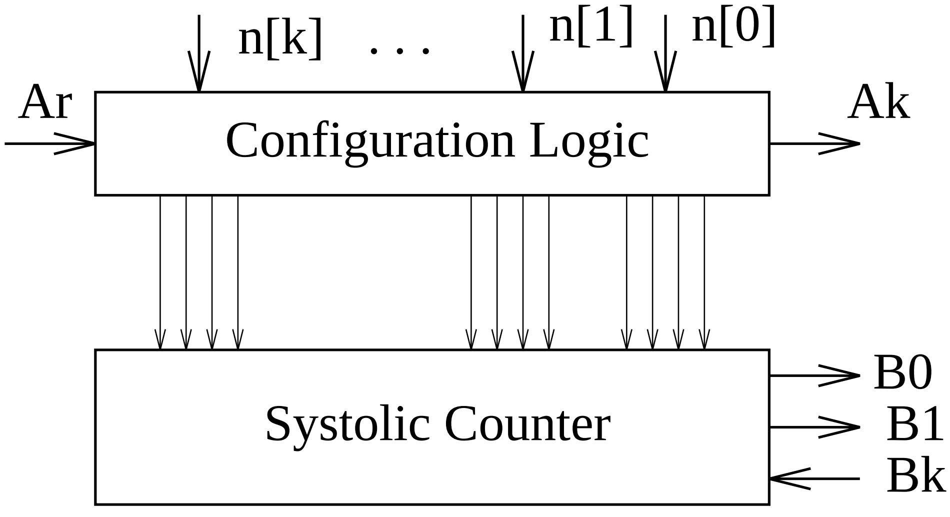
Counter for $n = 5$ at the Petri net level



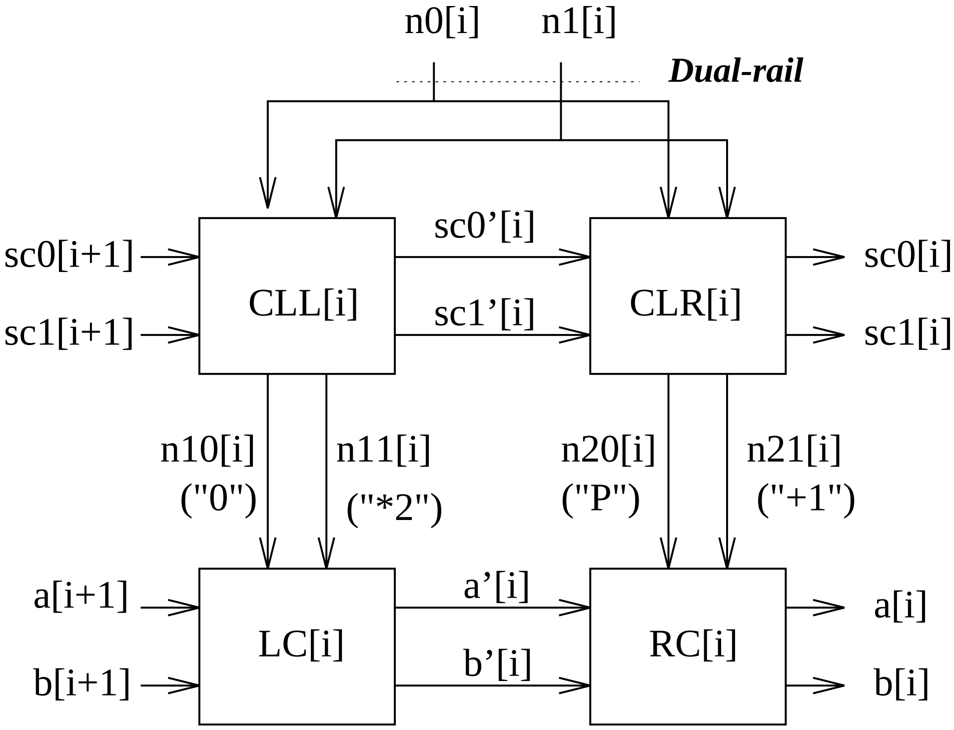
Counter for $n = 5$: Petri net composition unfolding



General structure



Bit-level structure



Cell Type Conditions (via Left Neighbour)

For $0 < i < k$:

Left Cell conditions:

$$LC[i] = \begin{cases} - & \text{if } \forall j \geq i : n[j] = 0 \\ 0 & \text{if } \forall j > i : n[j] = 0 \wedge n[i] = 1 \\ *2 & \text{if } \exists j > i : n[j] = 1 \end{cases} = \begin{cases} - & \text{if } RC[i+1] = \text{"-"} \ \wedge n[i] = 0 \\ 0 & \text{if } RC[i+1] = \text{"-"} \ \wedge n[i] = 1 \\ *2 & \text{if } RC[i+1] \neq \text{"-"} \end{cases}$$

Right Cell conditions:

$$RC[i] = \begin{cases} - & \text{if } \forall j \geq i : n[j] = 0 \\ P & \text{if } \exists j > i : n[j] = 1 \wedge n[i] = 0 \\ +1 & \text{if } n[i] = 1 \end{cases} = \begin{cases} - & \text{if } LC[i] = \text{"-"} \\ P & \text{if } LC[i] \neq \text{"-"} \ \wedge n[i] = 0 \\ +1 & \text{if } LC[i] \neq \text{"-"} \ \wedge n[i] = 1 \end{cases}$$

Cell Type Conditions (via Left Neighbour)

For $i = k$ (MSB):

Left Cell conditions:

$$LC[k] = \begin{cases} - & \text{if } n[k] = 0 \\ 0 & \text{if } n[k] = 1 \end{cases}$$

Right Cell conditions:

$$RC[k] = \begin{cases} - & \text{if } LC[k] = \text{"-"} \\ +1 & \text{if } LC[k] \neq \text{"-"} \end{cases}$$

For $i = 0$ (LSB):

Left Cell conditions:

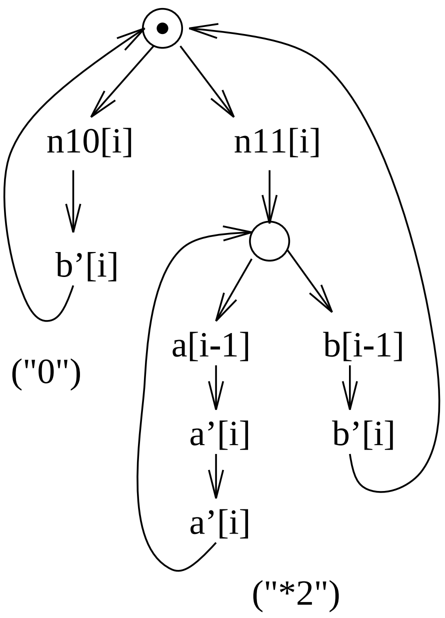
$$LC[0] = \begin{cases} 0 & \text{if } RC[1] = \text{"-"} \\ *2 & \text{if } RC[1] \neq \text{"-"} \end{cases}$$

Right Cell conditions:

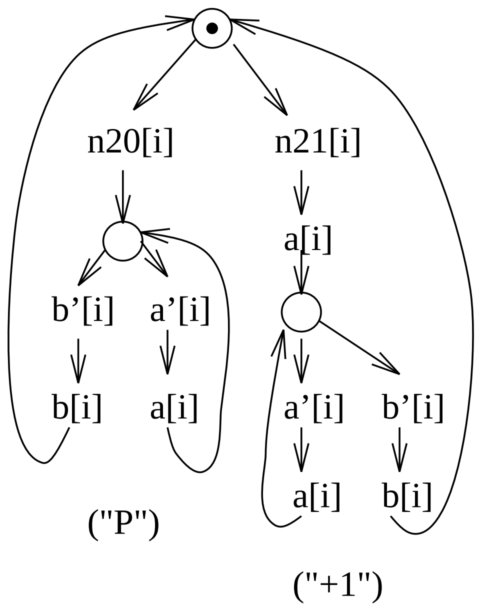
$$RC[0] = \begin{cases} P & \text{if } n[0] = 0 \\ +1 & \text{if } n[0] = 1 \end{cases}$$

Petri net models of $LC[i]$ and $RC[i]$

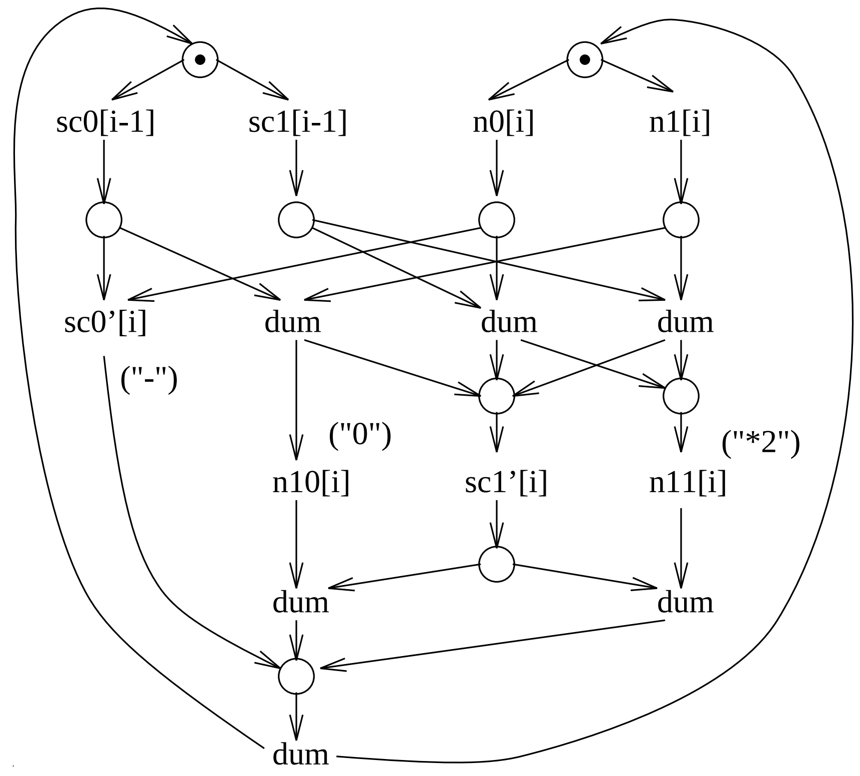
LC[i]



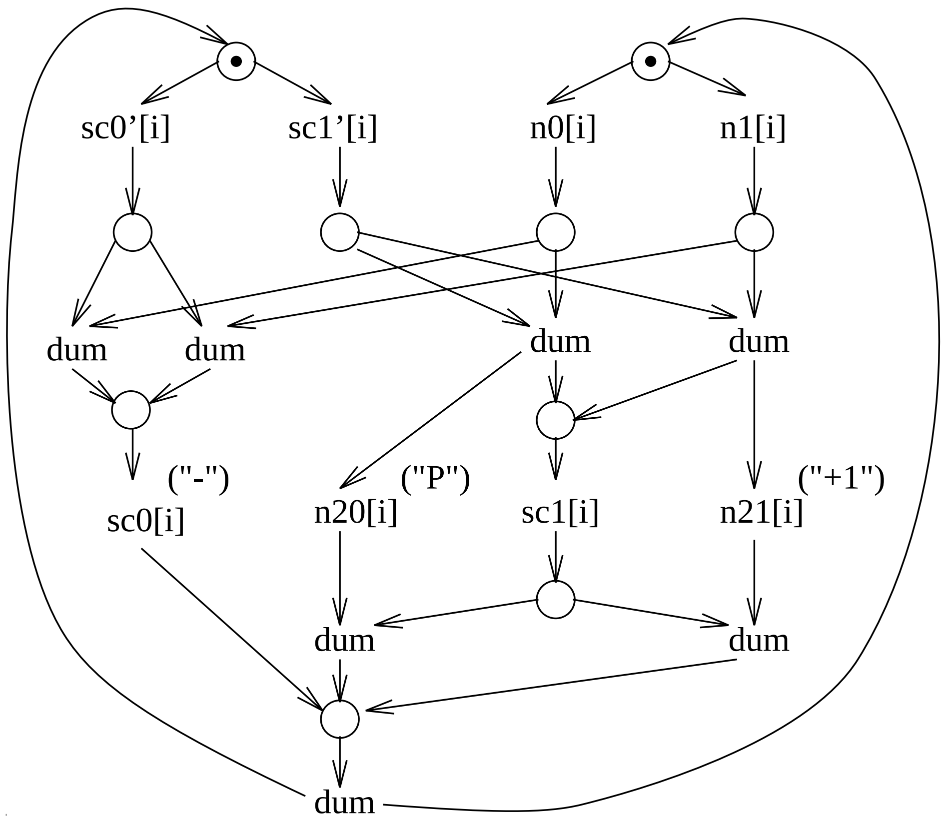
RC[i]



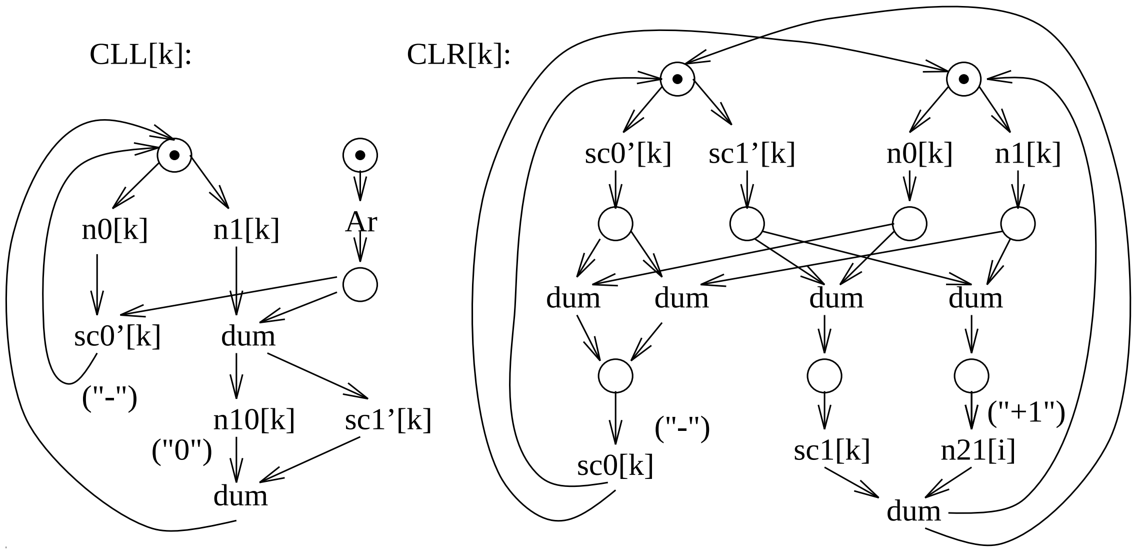
Petri net model of $CLL[i]$



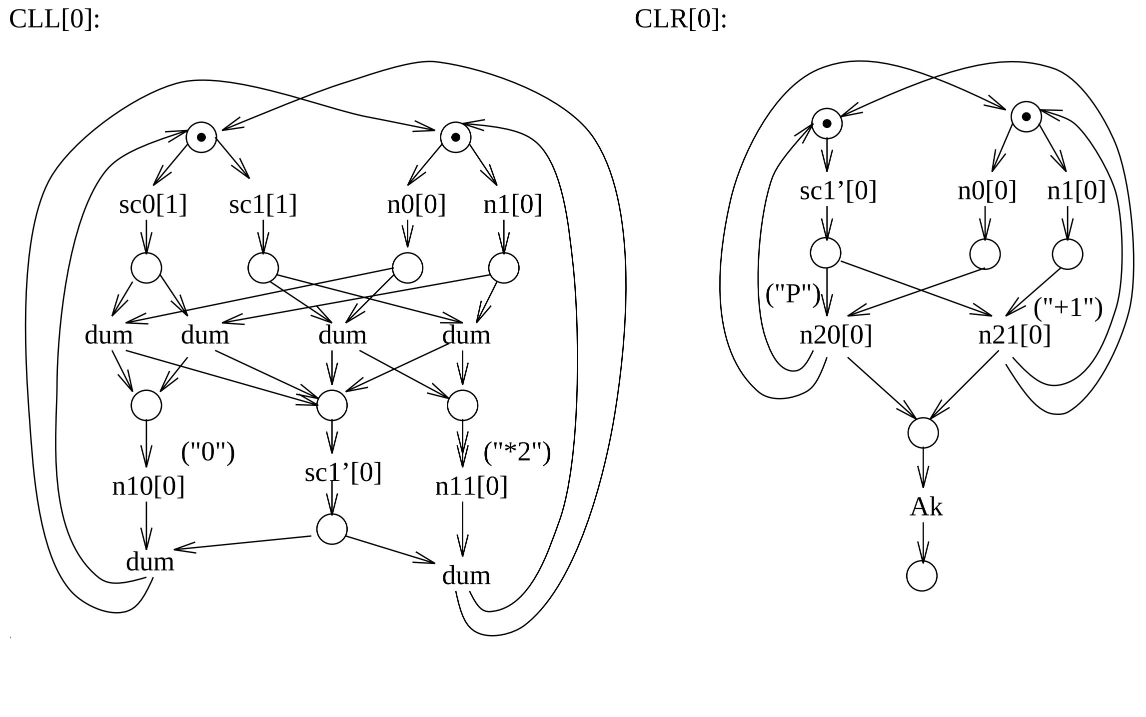
Petri net model of $CLR[i]$



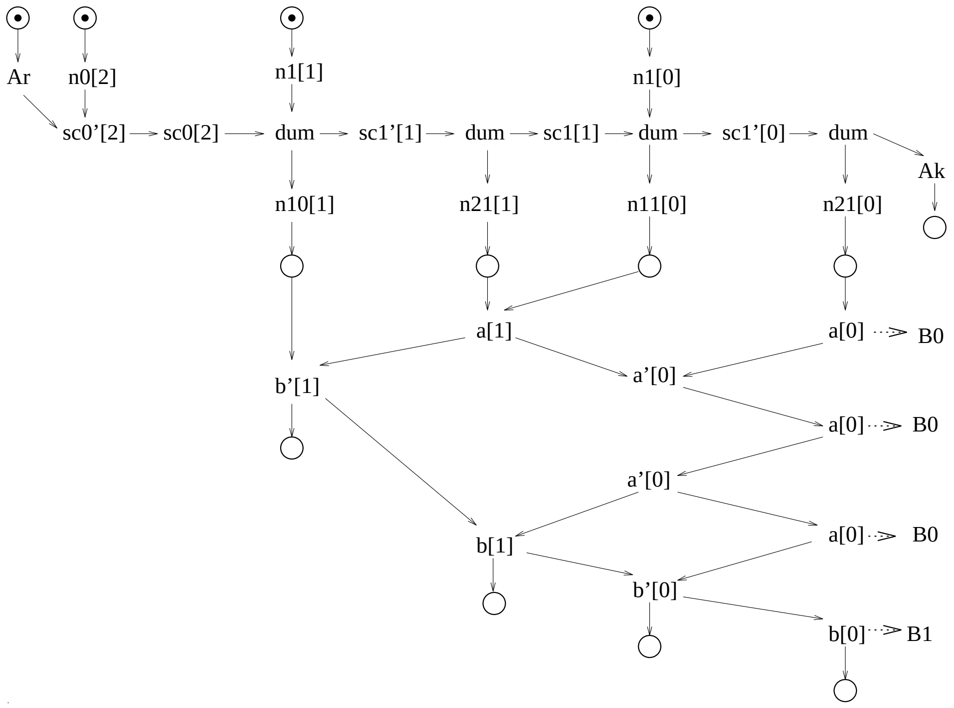
Petri net models of $CLL[k]$ and $CLR[k]$ (MSB)



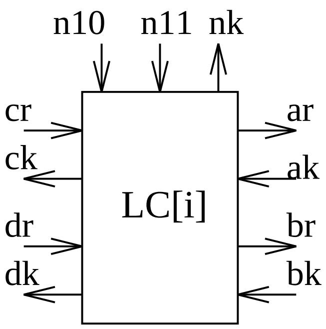
Petri net models of $CLL[0]$ and $CLR[0]$ (LSB)



Counter loaded with $n = 3$: Petri net unfolding

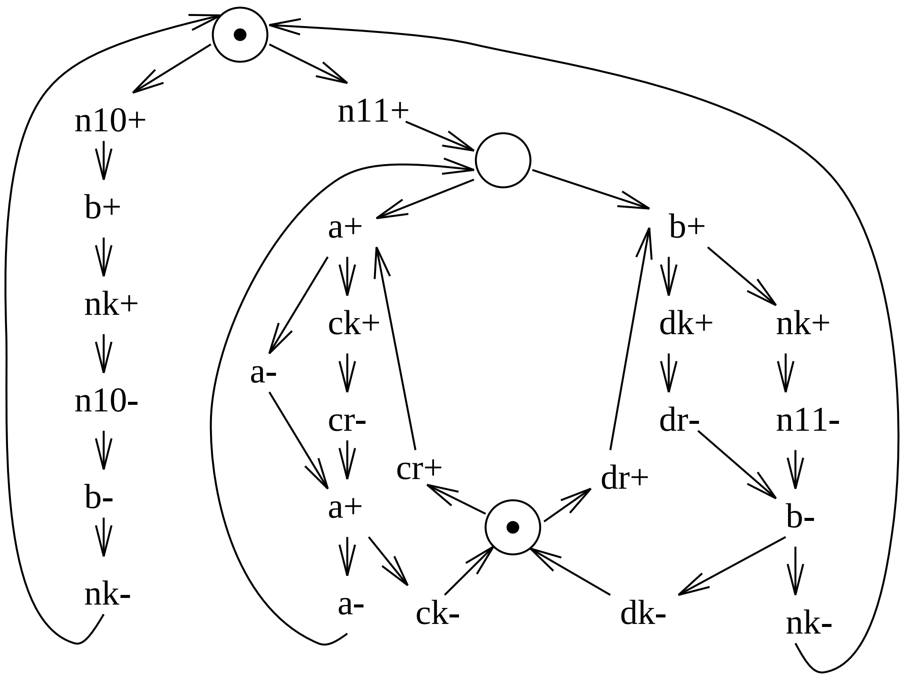


STG model of Left Cell $LC[i]$



For MSB LC[k]

use only part "0"

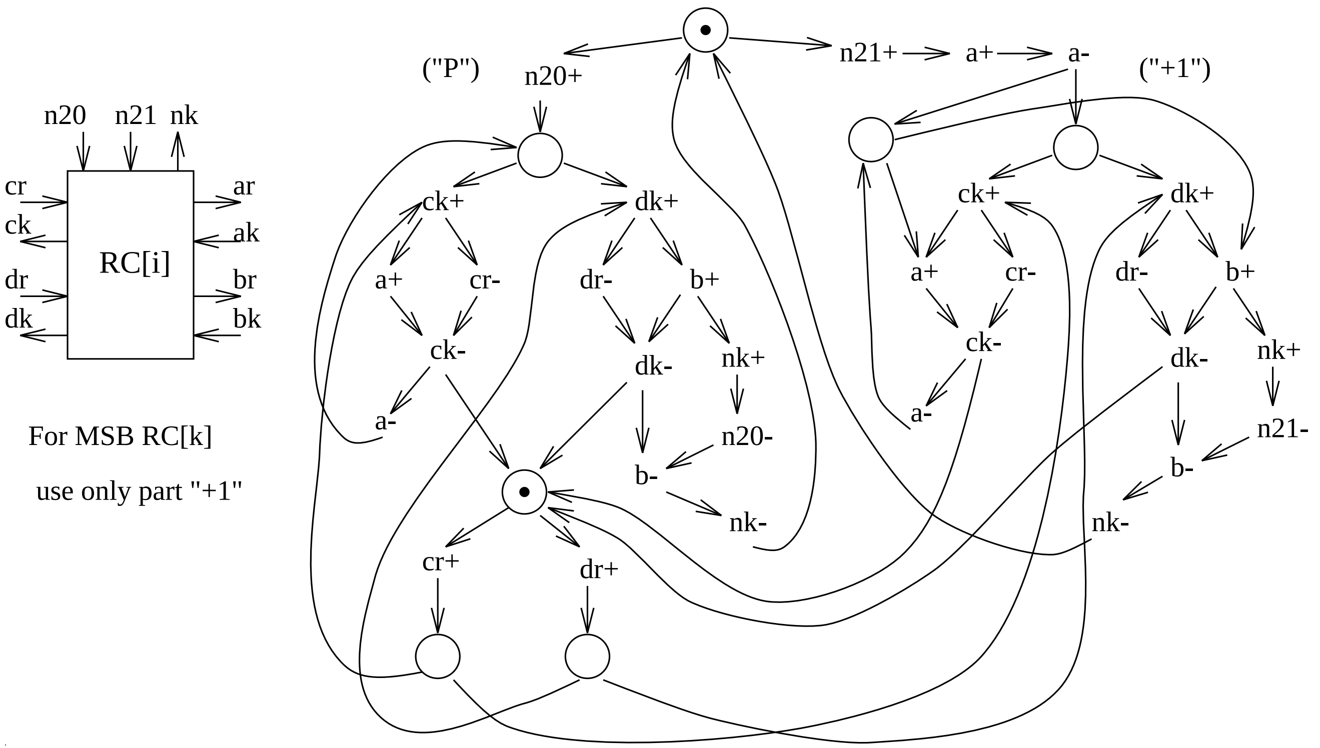


("0") ("*2")

Implementation of Left Cell $LC[i]$

```
ar = csc0 * csc1' + csc0'* csc1
br = !ak * dr * n11 + dk * dr + dk * n11 + n10
ck = ak * csc0 + !ak * csc1 + csc0 * csc1
dk = bk * dk + bk * n11
nk = bk * !n11 + dk
csc0 = ak * csc0 + !ak * cr * n11 + cr * csc0
csc1 = ak * csc0 + !ak * csc1 + csc0 * csc1
```

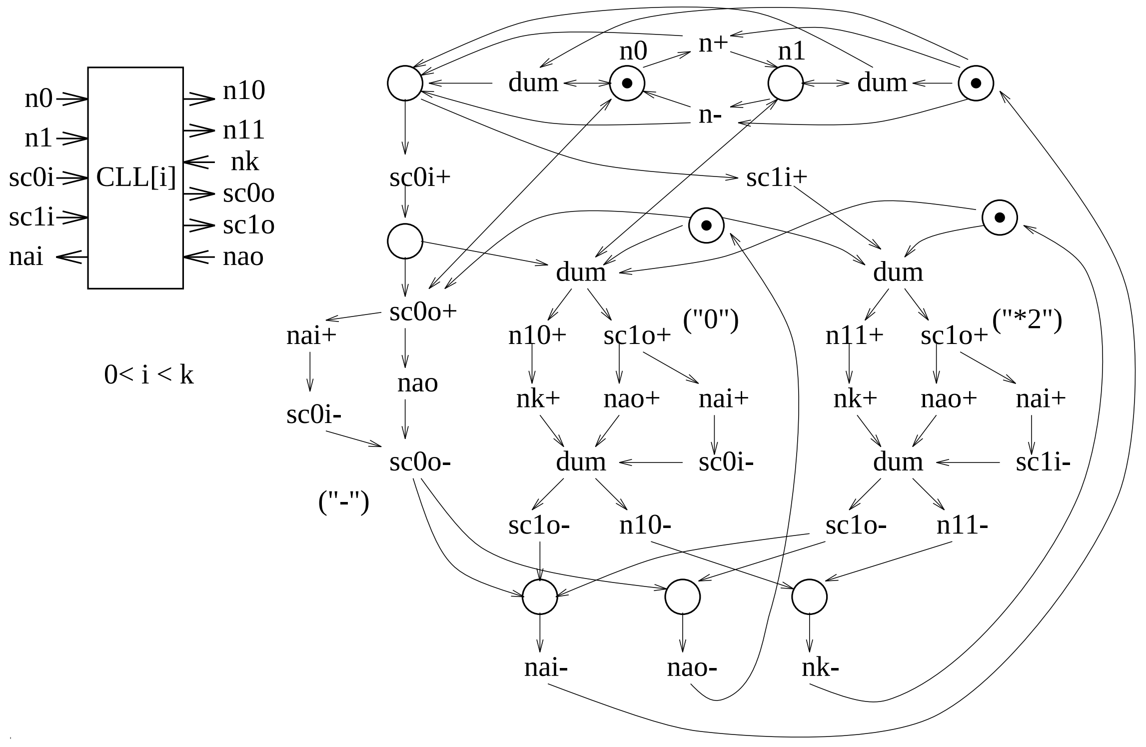

STG model of Right Cell $RC[i]$



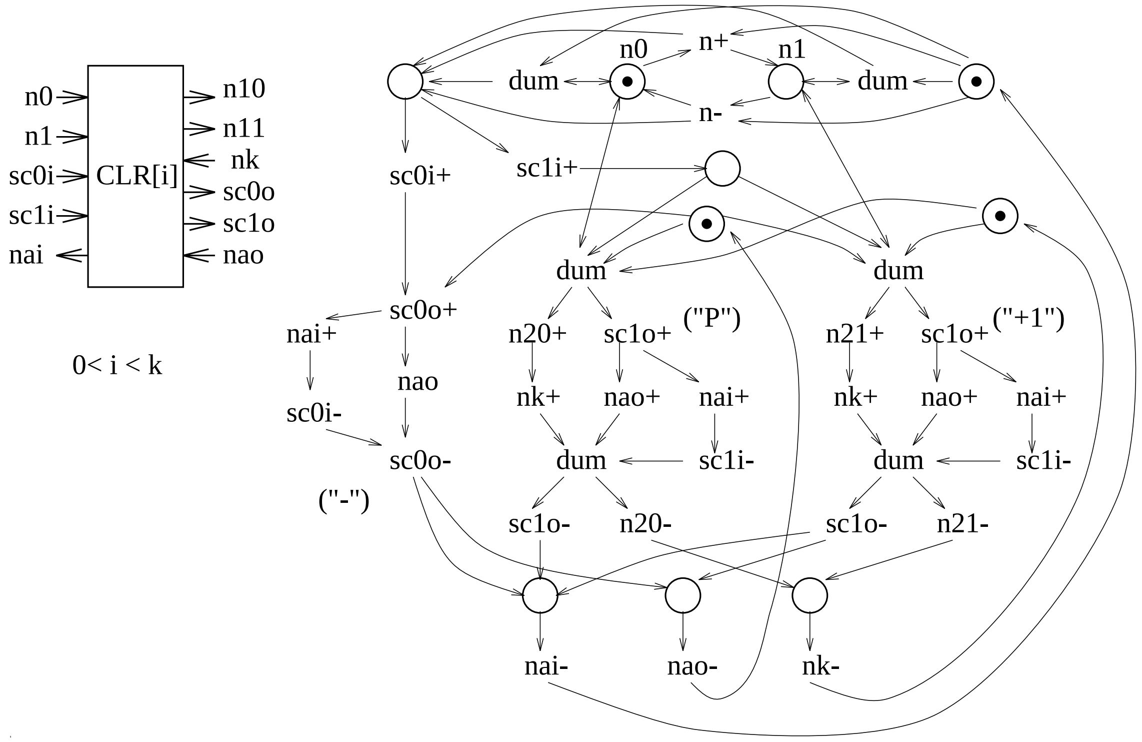
Implementation of Right Cell $RC[i]$

```
ar = !bk * !csc0 * n21 + ck
br = br * n20 + br * n21 + dk
ck = !ak * !bk * cr * csc0 + !ak * !bk * cr * n20
     + !ak * ck + ck * cr
dk = !ak * !bk * csc0 * dr + !ak * !bk * dr*  n20
     + !bk * dk + dk * dr
nk = bk * !csc0
csc0 = a * n21 + !bk csc0
```

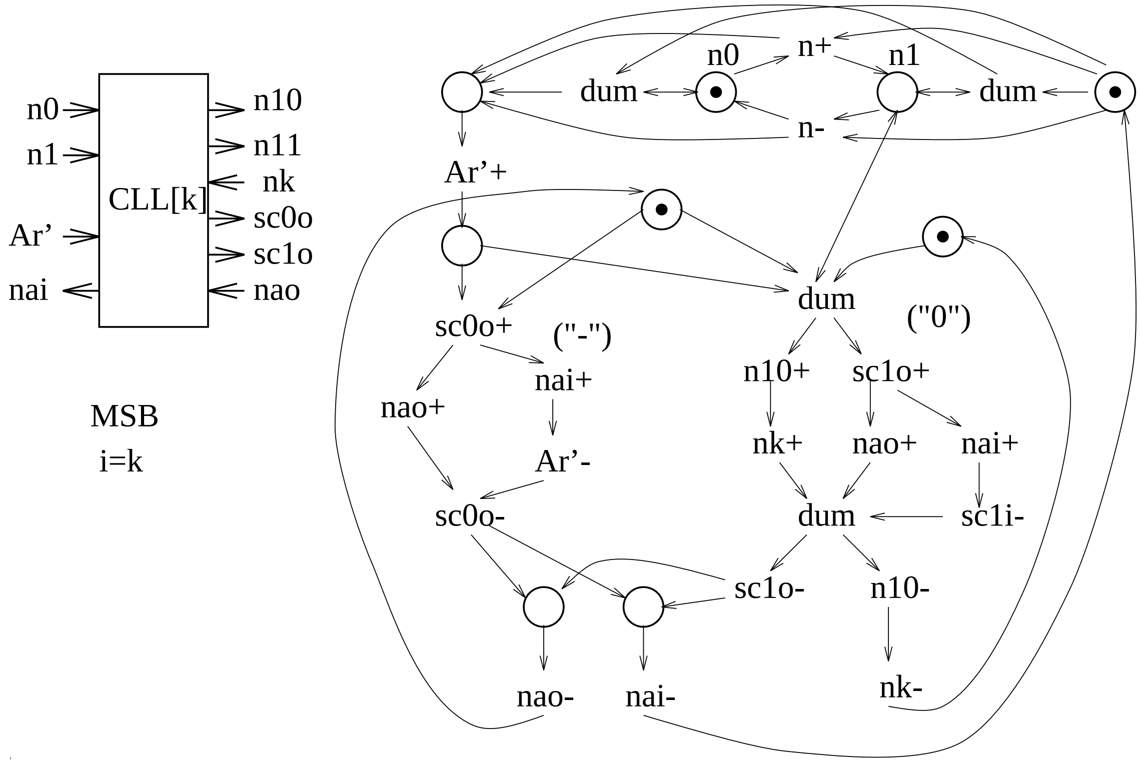

STG model of Left Cell Control $CCL[i]$



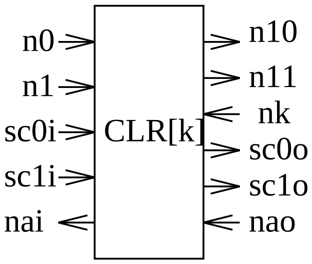
STG model of Right Cell Control $CCR[i]$



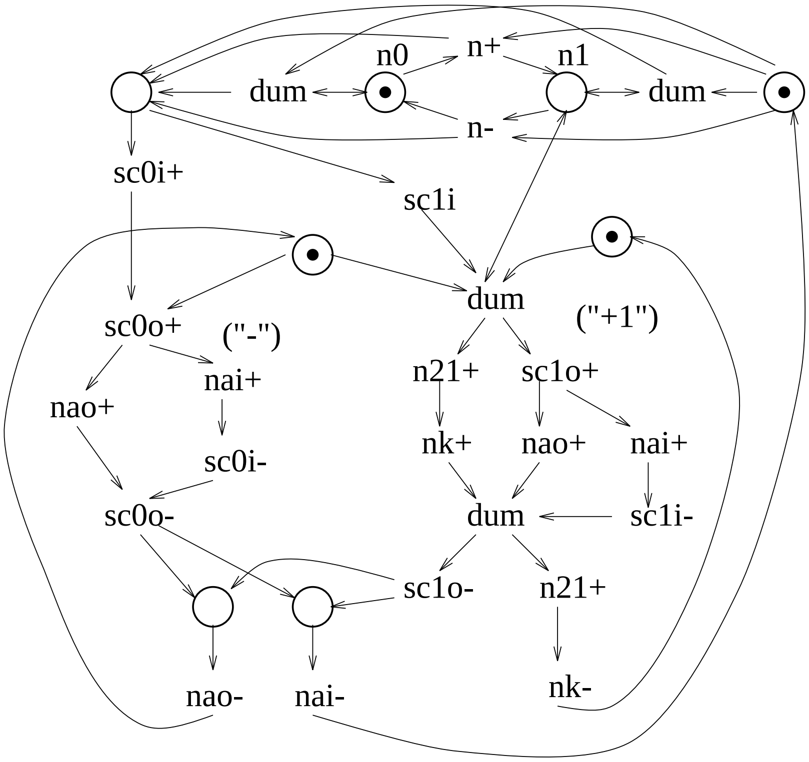
STG model of MSB Left Cell Control $CCL[k]$



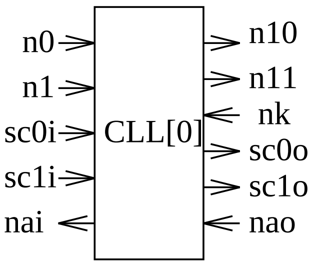
STG model of MSB Right Cell Control $CCR[k]$



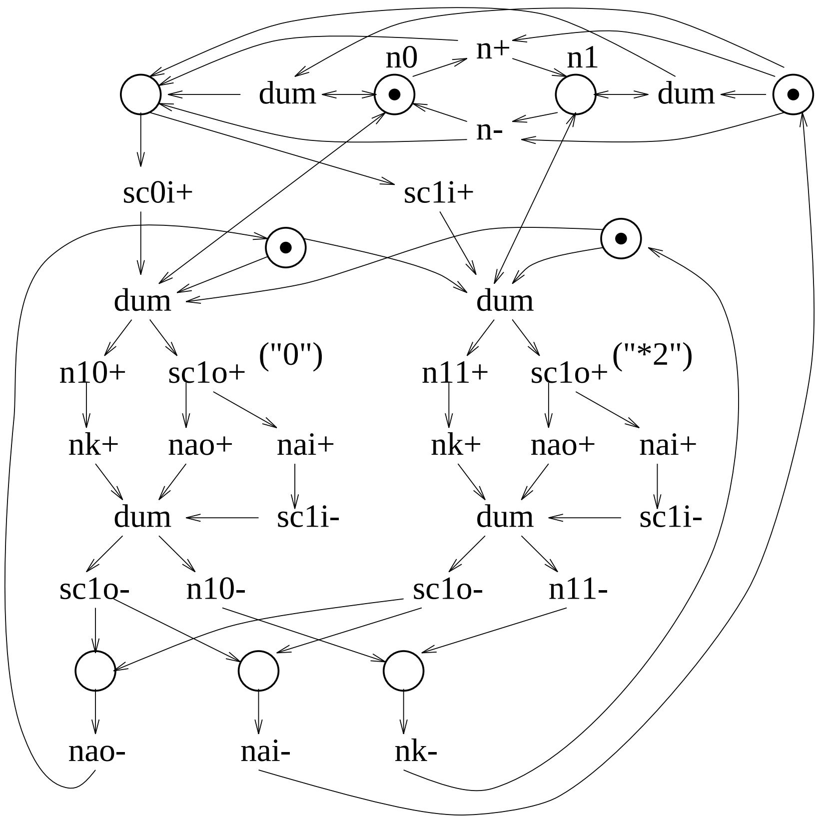
MSB
i=k



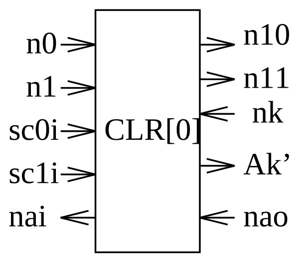
STG model of LSB Left Cell Control $CCL[k]$



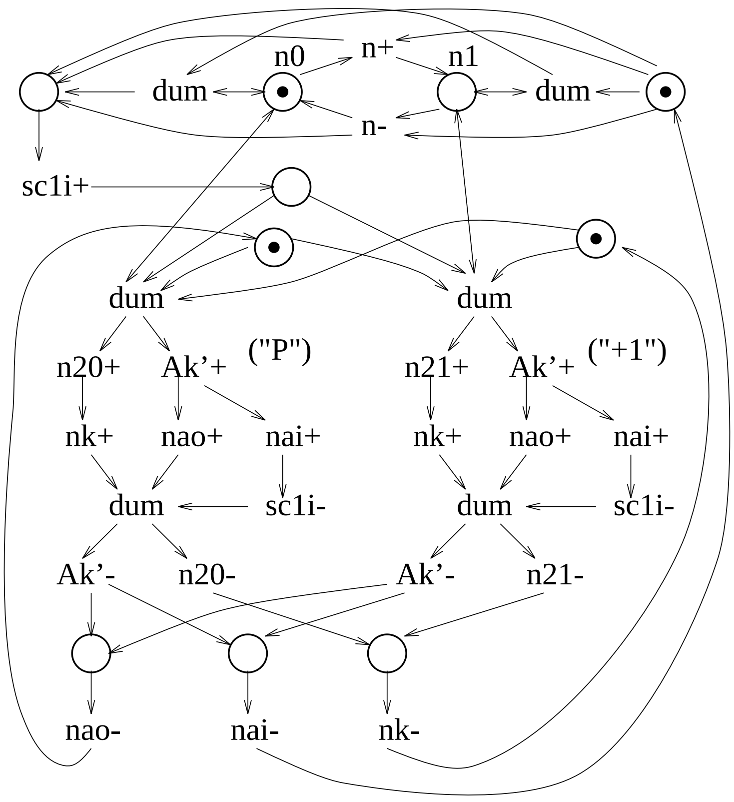
LSB
i=0



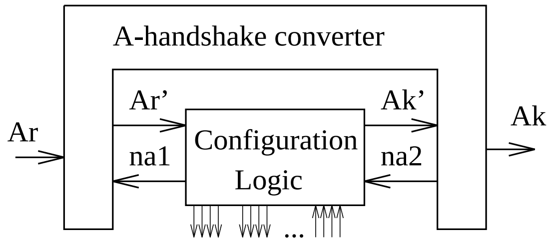
STG model of LSB Right Cell Control $CCR[k]$

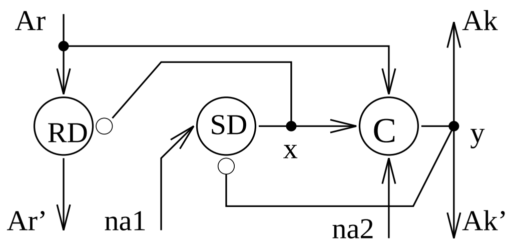


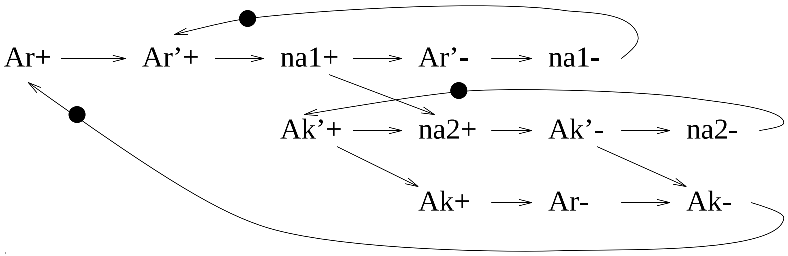
LSB
i=0



Handshake converter from (Ar, Ak) to pipeline







Reset-Dominant Latch:

Ar'=Ar*!x+Ar'*!x

Set-Dominant Latch:

x = na1+x*!y

On Performance of the Counter

- The original spec

$$(A?n; (B!0)^n; B!1)*$$

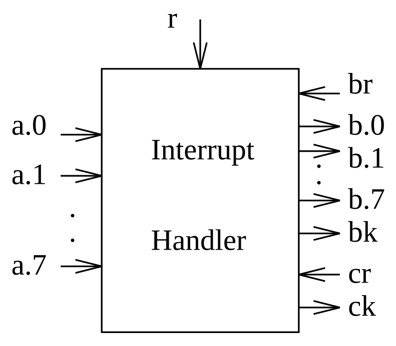
presumes that counter is loaded with n after *every* counting cycle

- This may introduce undesirable (for whom ?) *delay*

- The specfication can be "adjusted" to allow for faster (systolic) restart *if n doesn't change* between cycles

- The above design will support that option – only the "Handshake Converter" is to be redesigned

Interrupt Handler: Structural View



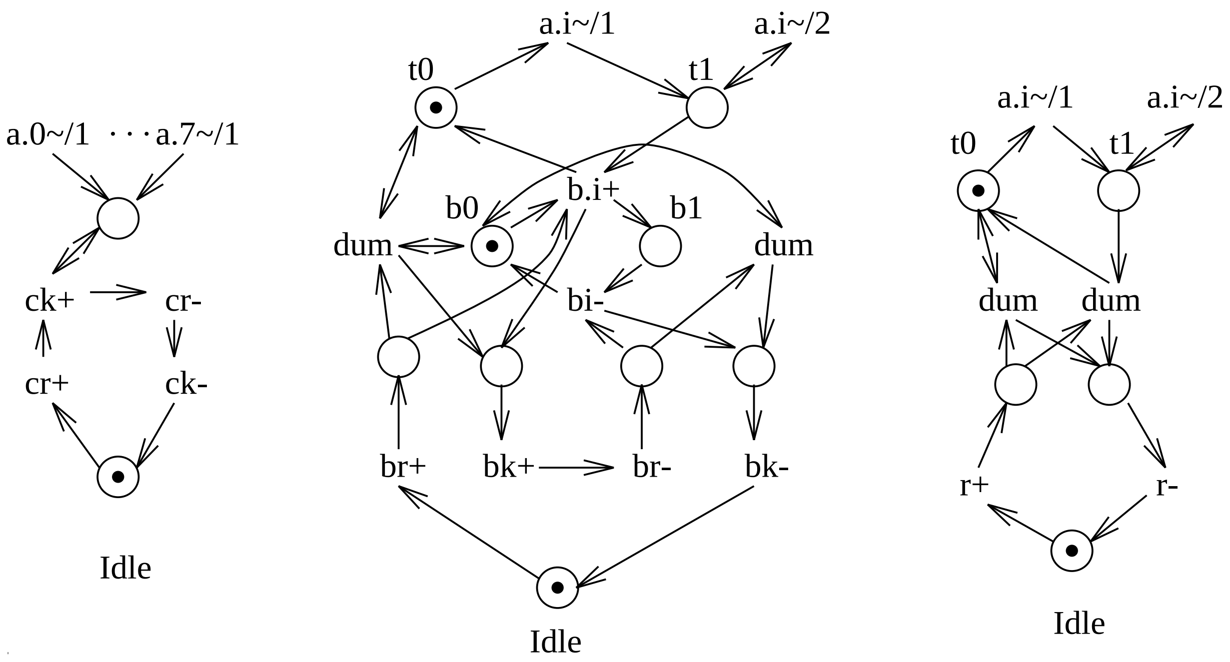
Passive Handshakes:
- (br, bk) for a b-command
- (cr, ck) for a c-command

Important Requirements:

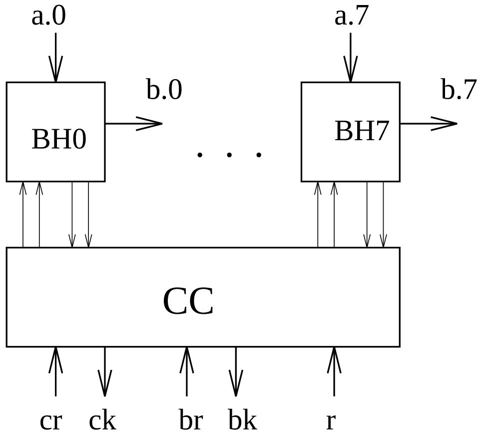
- during the data valid interval outputs $b.j$ must be stable whereas inputs $a.j$ may change

- low power option – no "busy wait" polling on a c command

Towards Petri Net Specification

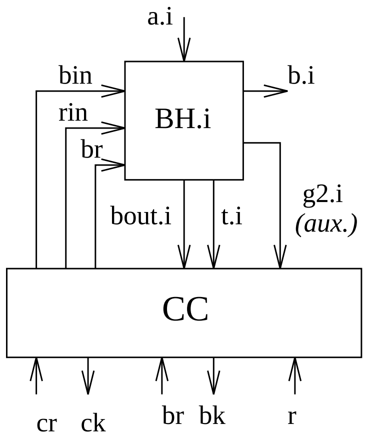


Overall Structure



- $BH0, \ldots, BH7$ are one-bit handlers
- CC is a coordination-synchronisation circuit

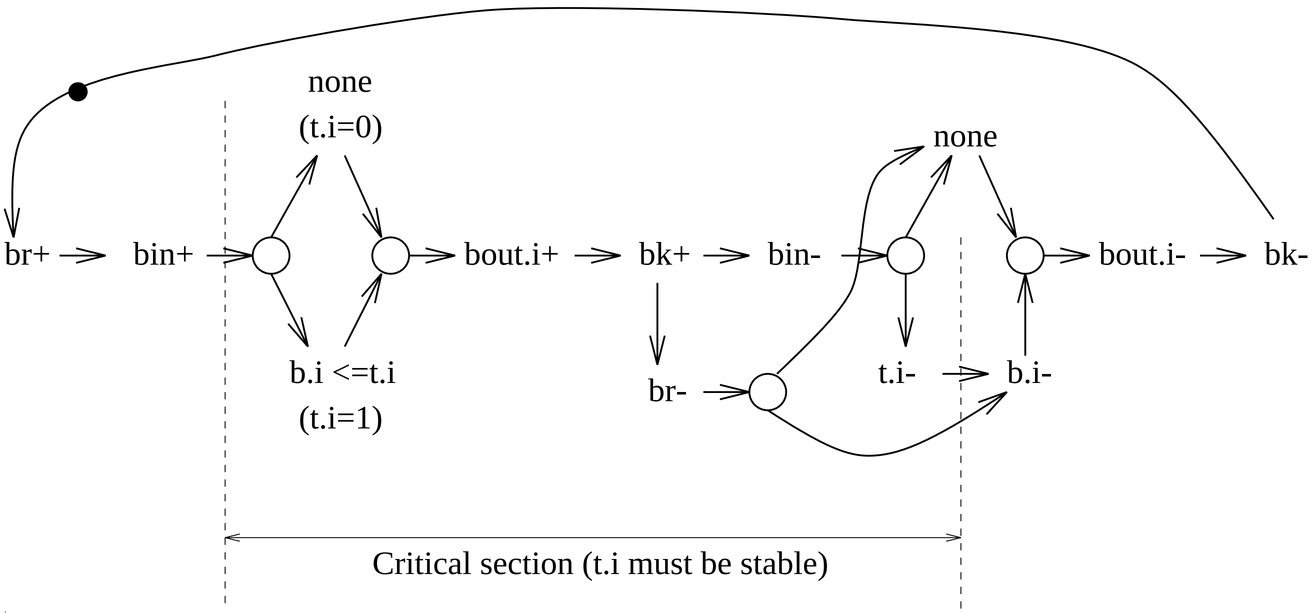
One-Bit Handler: Basic Structure



Basic Action:

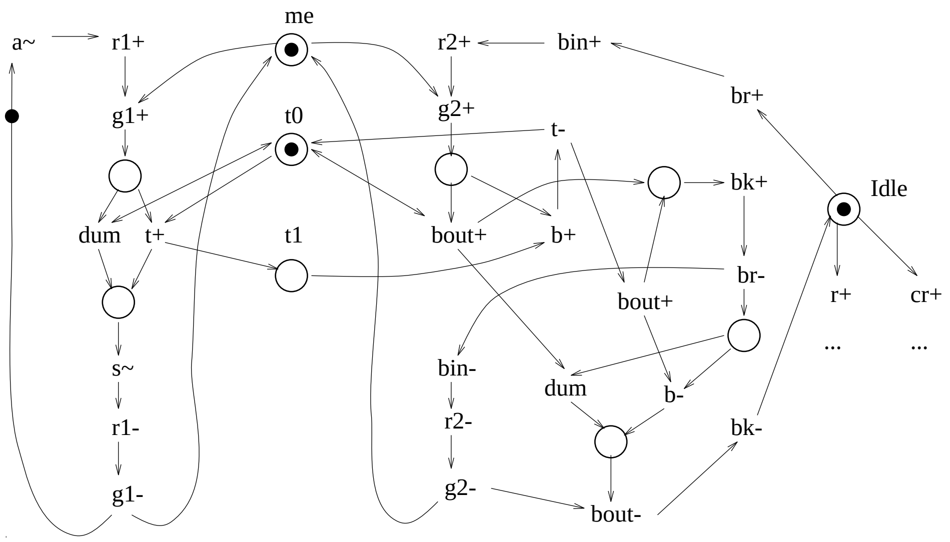
- Latch event on $a.i$ into $t.i$ before br or r
- Collect $t.i$ via OR logic to produce ck upon cr
- Rewrite $t.i$ into $b.i$ upon br, reset $t.i$

Basic Schedule for the execution of command b



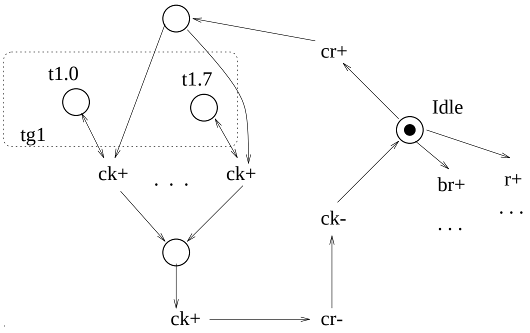
The critical section must be organised via *mutual exclusion* between latching events on $a.i$ into $t.i$ and processing b

Basic STG for command b



Timing assumption: The minimal interval between two adjacent changes of interupt signal $a.i$ should be at least equal to the delay of the *Mutex* element and latch for $t.i$

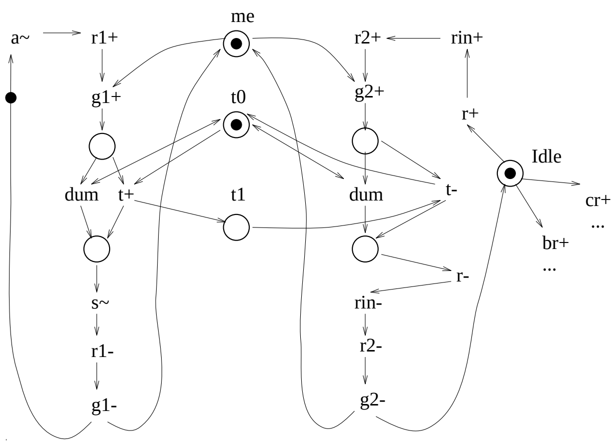
Basic STG for command c



Important:

- Signal $c.k$ is produced as soon as at least one of $t.i$ is set to 1 – this requires the gloabal signal $tg = t.0 + \ldots + t.7$.
- Ack-ing signal tg may be a problem for pure speed-independence (tg must be included into the STG specification).

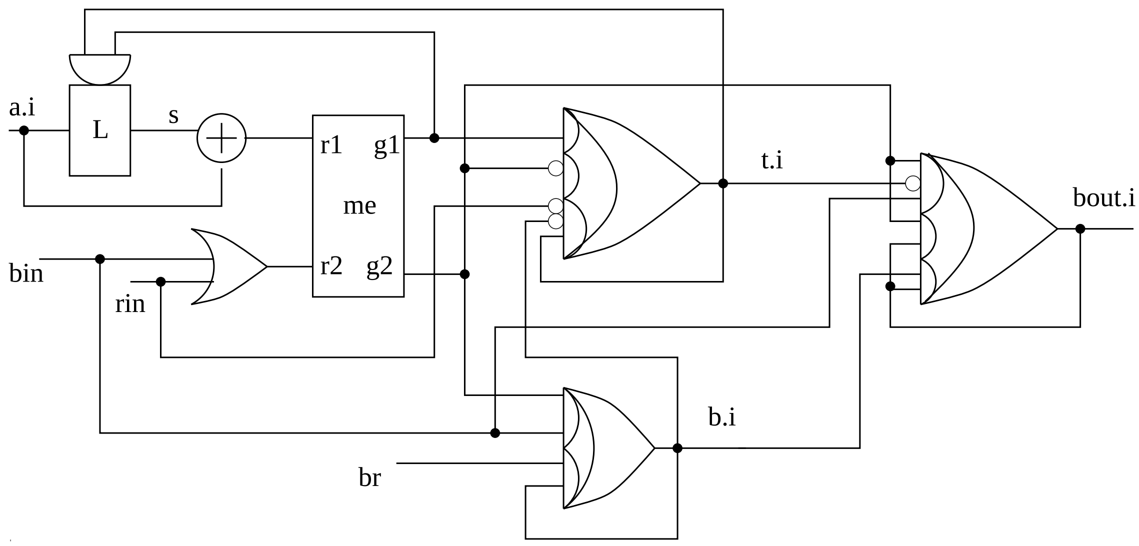
Basic STG for command r



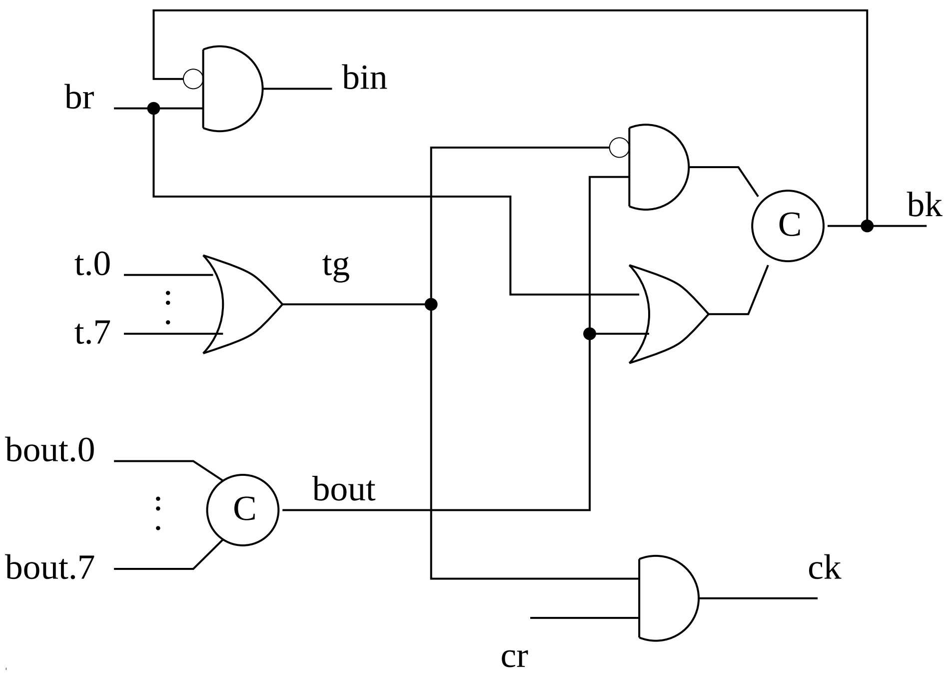
Timing assumption:

- The minimal interval between $r+$ and $r-$ should be sufficient for the circuit to reset its latches $t.i$ (including *Mutex* delays.
- Internally we may implement handshake rin and $rout$ (similar to bin and $bout$)

Implementation for One-Bit Handler



Implementation for Cordination Circuit



This circuit is not speed-independent wrt OR gate for tg

Speed-Independent Design (for two bit handlers)

Output from petrify:

```
[bk] =  bout*g2t + br*bk;
[ck] =  cr*tg;
[t1] =  !g12*t1 + g11 + t1*bin;
[t2] =  !g22*t2 + g21 + t2*bin;
[tg] =  t1 + t2;
[b1] =  br*b1 + tg*b1 + !r*g12*t1;
[b2] =  br*b2 + tg*b2 + !r*g22*t2;
[bin] =  !bk*bin + br*!bk*!g2t;
[bout1] =  g12*!t1*bin + b1;
[r11] =  a1*!s1 + !a1*s1 = XOR(a1,s1);
[r21] =  a2*!s2 + !a2*s2 = XOR(a2,s2);
[r12] =  r + bout + bin;
[r22] =  r + bout + bin;
[bout2] =  g22*!t2*bin + b2;
[bout] =  bout1*bout + bout1*bout2 + bout2*bout;
[g2t] =  g22*g2t + g12*g22 + g12*g2t = C(g12,g22);
[s1] =  a1*g11*t1*tg + !tg*s1 + !g11*s1 + !t1*s1;
[s2] =  a2*g21*t2*tg + !tg*s2 + !g21*s2 + !t2*s2;
```

Speed-Independent Design with C-gates (for two bit handlers)

Output from petrify:

```
[bk_C1] =  bout*g2t; [bk_C2] =  br;
[ck] =  cr*tg;
[t1_C1] =  g11; [t1_C2] =  !t1 + !r12 + !g2t + bin;
[t2_C1] =  g21; [t2_C2] =  !t2 + !r22 + !g2t + bin;
[tg] =  t1 + t2;
[b1_C1] =  g12*t1*bin; [b1_C2] =  !b1 + br + t1;
[b2_C1] =  g22*t2*bin; [b2_C2] =  !b2 + br + t2;
[bin_C1] =  br*!bk*!g2t; [bin_C2] =  !bk;
[bout1] =  b1 + g12*!t1*bin;
[r11] =  !a1*s1 + a1*!s1;
[r12] =  r + bout + bin;
[bout2] =  b2 + g22*!t2*bin;
[bout_aux_C1] = bout1; [bout_aux_C2] = bout2
[bout] = bout_aux + bout * tg;
[r21] =  !a2*s2 + a2*!s2;
[r22] =  r + bout + bin;
[g2t_C1] =  g22; [g2t_C2] =  g12;
[s1_C1] =  a1*g11*t1*tg; [s1_C2] =  !g11 + !t1 + !tg + a1;
[s2_C1] =  a2*g21*t2*tg; [s2_C2] =  !g21 + !t2 + !tg + a2;
```


Remarks on Speed-Independence

- Transitions of the OR-gate for tg must be ack-ed in both b and r commands
- Neither of BIH is allowed to leave the critical region until ALL of them completed it – hence synchronisation of $g2$ on $g2t$
- The last solution allows resetting of the Mutex'es on $(r2, g2)$ only after all $b.i$ are reset – this means a slight delay in latching events on $a.i$ into $t.i$ (the latter waits for release of Mutex).

Conclusions

- Petri nets allow modelling the design at various levels of abstraction – from high level labelled actions to STG-level
- The STGs (up to 30 signals) can be implemented by SIS and/or petrify (the latter can also solve Complete State Coding)
- It is (increasingly) possible to have a limited technology mapping within petrify
- New tools (using approximation-refinement techniques) are needed to deal with larger STGs – unfoldings ?
- Interested in more solutions/discussion? Please attend the talk!